Assessing Survivability Using Software Fault Injection

Jeffrey Voas
Reliable Software Technologies
21351 Ridgetop Circle, #400
Dulles, VA 20166
jmvoas@rstcorp.com

Abstract

In this paper, we present an approach and experimental results from using software fault injection to assess information survivability. We define information survivability to mean the ability of an information system to continue to operate in the presence of faults, anomalous system behavior, or malicious attack. In the past, finding and removing software flaws has traditionally been the realm of software testing. Software testing has largely concerned itself with ensuring that software behaves correctly — an intractable problem for any non-trivial piece of software. In this paper, we present "off-nominal" testing techniques, which are not concerned with the correctness of the software, but with the survivability of the software in the face of anomalous events and malicious attack. Where software testing is focused on ensuring that the software computes the specified function correctly, we are concerned that the software continues to operate in the presence of faults, unusual system events or malicious attacks.

1 Introduction

Our motivation for researching advanced software assessment techniques fits in line with the following comments made by the committee that wrote the 1998 *Trust in Cyberspace* report:

1. "The absence of standard metrics and a recognized organization to conduct assessments of trustworthiness is an important contributing factor to the problem of imperfect information. In some industries, such as pharmaceuticals, regulatory mandate has resolved this problem by requiring the development and disclosure of information."

2. "A consumer may not be able to assess accurately whether a particular drug is safe but can be reasonably confident that drugs obtained from approved sources have the endorsement of the US Food and Drug Administration (FDA) which confers important safety information. Computer system trustworthiness has nothing comparable to the FDA. The problem is both the absence of standard metrics and a generally accepted organization that could conduct such assessments. There is no Consumer Reports for [software and information] Trustworthiness."

These statements highlight two key problems facing software users and consumers alike: (1) a lack of sound metrics for quantifying that information systems are trustworthy, and (2) the absence of an organization (such as an Underwriter's Laboratory) to apply the metrics in order to assess trustworthiness. In fact, if these problems were solved, software vendors who sought to provide reliable products would also benefit.

Note, however, that these two problems are not of equal size. Problem (1) is the more difficult and problem (2) can be achieved more easily, but only after problem (1) is solved.

The lack of sound, fair, and quantitative metrics for software safety, reliability, security, and fault-tolerance have contributed to the distrust of Cyberspace mentioned in the report. There is a deeper problem here however, and that is that software quality is more difficult to assess than it is to achieve. This problem is unique to software; physical systems do not experience it. For example, it is far easier to determine if a ball bearing has been perfectly manufactured via an electron microscope than it is to produce perfect ball bearings. Such a situation is not true for software.

Our software research projects over the last 4 years have focused on creating automated technologies and metrics to assess software trustworthiness. Our belief is that enough emphasis has been applied to process improvement methods to improve software quality (even though those processes are often ignored). If

Paper presented at the RTO IST Symposium on "New Information Processing Techniques for Military Systems", held in Istanbul, Turkey, 9-11 October 2000, and published in RTO MP-049.

we can better assess the quality of software systems, then hopefully the distrust can be reduced and as a side-benefit, we will be able to assess the return-on-investment from software process improvement.

We acknowledge, along with the report, that the US Government has not ignored the software assessment problem. They have invested heavily in software testing research for the past 20 years. Software testing is still the most common approach for determining whether software will behave as desired. Unfortunately, however, the outcome of that research is not applicable to the large-scale survivability problems endemic to the Internet.

As noted in the *Trust in Cyberspace* report, this research has focused more on testing "in the small" than testing "in the large." While this enables better subsystems, it does not address the interaction problems that weaken survivability:

> "Much of the research in testing has been directed at dealing with problems of scale. The goal has been to maximize the knowledge gained about a component or subsystem while minimizing the number of test cases required. Approaches based on statistical sampling of the input space have been shown to be infeasible if the goal is to demonstrate ultra-high levels of dependability [5], and approaches based on coverage measures do not provide quantification of useful metrics such as mean time to failure. The result is that, in industry, testing is all too often defined to be complete when budget limits are reached, arbitrary milestones are passed, or defect detection rates drop below some threshold. There is clearly room for research - especially to deal with the new complications that MIS brings to the problem: uncontrollable and unobservable subsystems."

Therefore research is needed to increase the observability of "ilities" such as safety, security, reliability, and survivability. In this paper we describe two areas of research that use off-nominal testing for survivability.

2 Off-Nominal Testing for Survivability

In this paper, we present an approach and experimental results from using software fault injection to assess information survivability. We define information survivability to mean the ability of an information system to continue to operate in the presence of faults, anomalous system behavior, or malicious attack. In the past, finding and removing software flaws has traditionally been the realm of software testing. Software testing has largely concerned itself with ensuring that software behaves correctly — an intractable problem for any non-trivial piece of software. In this paper, we present "off-nominal" testing techniques, which are not concerned with the correctness of the software, but with the survivability of the software in the face of anomalous events and malicious attack. Where software testing is focused on ensuring that the software computes the specified function correctly, we are concerned that the software continues to operate in the presence of faults, unusual system events or malicious attacks.

The off-nominal testing approach uses fault injection analysis to determine how survivable a program is to unusual events that can occur during field operation. Fault injection is the process of perturbing program behavior by corrupting a program state during program execution. Corrupting program states can affect program control flow as well as corrupt program data. We use fault injection analysis to assess information survivability under three different scenarios:

- software flaws in program source code,
- malicious attacks against programs,
- anomalous behavior from third party software.

To assess the survivability of a program, we must know how robust it is under flawed software conditions. Since most programs today contain on average one defect for every 6000 lines of source code, we know that today's systems are deployed with a great number of undiscovered software flaws that may be triggered in the field at anytime [8]. If we knew *a priori* where these flaws exist, we would be able to locate and fix them. However, since we do not know where these flaws are, we simulate their effects by automatically corrupting program state at as many program locations as possible and assessing the effect on survivability of a program state corruption at a particular location. The effect on security and safety of software flaws has been documented in great detail in BugTraq[1] and in [7].

The technique to simulate software flaws uses program state corruption. Since, the range of possible effects on program state is too great to use specific program corruptions, we use random program corruptions

[1] See www.securityfocus.com for BugTraq archives.

for specific program state types. For instance, we can corrupt program memory by using random number selection based on the program data type. Program control flow can be corrupted by corrupting Boolean conditions in control flow constructs.

In the second scenario, we are interested in assessing the impact of malicious attacks against programs. In this scenario we can use directed fault injection techniques that subject a software program to the types of well-known attacks it may experience in the field. The most common attack by far is the buffer overrun attack. We have developed specific fault injection functions to test the vulnerability of program buffers to "stack-smashing" buffer overrun attacks. On occasion, testing using random program state corruption to simulate software flaws will sometimes result in unveiling a security flaw. Examples of using these techniques against commonly used network servers are presented later in this paper.

Finally, we are interested in assessing the impact of failing third party software on information survivability. This topic is important to gauge survivability of an information system because today's software is almost always built using third party software such as libraries and commercial off-the-shelf (COTS) components. In the preceding two analyses, we use the source code of the program to perform the fault injection analysis. In assessing the impact of third party software failures on system survivability, we cannot assume access to source code for the third party software (such as proprietary operating system code or COTS software components). As a result, we have developed a technique we call Interface Propagation Analysis (IPA) that gives us the ability to assess the impact of failing third party software in the system under consideration. It is briefly described in Section 4.

3 Source-Code–Based Fault Injection

Fault injection can be applied to software source code by inserting instrumentation "hooks" into the original program source. The idea is to be able to observe program state and corrupt either control flow or data flow at particular locations within the source code. By corrupting program state, we can assess the impact on system survivability to inadvertent flaws or deliberate attacks against the program.

In the fault-error-failure model of software, a fault is introduced by a programmer, known as a "bug" in common parlance. The fault may be an error in the design of an algorithm or a simple coding error, such as an unconstrained buffer array. The fault is innocuous until it is activated (or triggered) by some input. At this point, the error is manifest. An error is only manifest when the resulting program state is incorrect (according to some correct specification) based the preceding program state and the current input. In other words, if the program state is correct, then the error is not manifest and the fault is inconsequential for the moment. Once the error is manifest, the program, or more generally, the system may continue to perform correctly or it may fail. If the system continues to perform correctly (or at least acceptably), then the error is either latent or it has been masked. If the system fails due to the error, then the error has been manifested as a failure.

We use fault injection to manifest errors. Thus, we are not introducing true faults in the fault-error-failure model sense; rather, we are injecting errors. A closer match to fault injection in the sense of the fault-error-failure model is mutation testing, where program code is selectively "mutated" or altered in order to determine if test cases can distinguish between good and flawed code [3]. Since we cannot know *a priori* where all program faults are, we manifest program errors by corrupting program states. If the errors we introduce during fault injection analysis cause system failure, then we have a measure for how survivable the

3.1 Implementation approaches for fault injection

The hypothesized errors that software fault injection uses are created by either: (1) adding code to the code under analysis, (2) changing the code that is there, or (3) deleting code from the code under analysis. One key requirement from these processes, however, is that the code that is either added, modified, or deleted must change either the software's output or an internal program state for at least one software test case. (Different applications of software fault injection will guide the decisions as to which of these two alternatives applies.) Without this requirement, the hypothesized errors will have had no semantic impact to the original code base and thus were meaningless (they were not anomalies at all). In mutation testing (a type of fault injection that we will discuss later), this is the dreaded "equivalent mutant" problem. The difficulty stems from the fact that equivalent mutants are often undetectable, forcing the costs to perform mutation testing to be much greater than they should be [9].

Figure 1 shows the software fault injection process. Code that is added to the program for the purpose of either simulating errors or detecting the effects of those errors is called *instrumentation code*. To perform fault injection, some amount of instrumentation is always necessary, and although this can be added

manually, it is usually performed by a tool. Instrumentation code can be placed on top of input or output interfaces to the software or directly into the logic of the software.

Instrumentation can be added into a variety of code formats: source code, assembly code, binary object code, etc. In short, any code format that can be compiled, interpreted, or that is ready for execution can be instrumented.

There are two key approaches for simulating errors: (1) directly changing the code that exists (this is referred to as code *mutation*), or (2) modifying the internal state of the program as it executes. We will now walk through an example of each approach beginning with code mutation.

Suppose a program has the following code statement:

```
a = a + 1;
```

This statement could be mutated as follows:

```
a = a + a + 1;
```

(provided that a does not have the value of zero). We could also modify the statement to:

```
a = a + 10;
```

And we could delete the statement as well. Note that all of these mutations change the resulting value of a from what it would have had not we not mutated the code (and for every test case that allows this statement to be executed).

The concept of forcefully changing the internal state of an executing program is a slight variation on the code mutation examples just shown. Clearly, each of the mutations above will change the state of the program after they are executed. But note that that is not necessarily true for all mutants. There are code mutants that although they are exercised will not modify the software's internal state. That would be the case if the value of a before the mutant a = a + a + 1 was executed is zero. (This would be an example of a transient fault using the definitions provided by Carriera *et al.*)

To forcefully modify a program's internal state to a value different than the one it currently has, we will add a function call to the code that overwrites the current internal value of a portion of the program's state. Typically, we overwrite programmer defined variables or the data that is being passed to or from function calls. By modifying this data, we are simulating the internal effects of faulty logic or any other anomalous event that could possibly affect the software's internal state.

The function calls we add to overwrite internal program values are termed *perturbation functions.* Perturbation functions are code instrumentation. When perturbation functions are applied to programmer defined variables, they typically either: (1) change the value of the variable to a value based on the current value, or (2) they pick a new value at random (independent of the original value). Also, they can simply return a constant replacement value if it is suspected that any fault placed at that point in the code would likely result in one particular value regardless of what the current value was. When non-constant replacement values are used, the perturbation function will produce random values based on the current value and a *perturbing distribution.* Non-constant perturbing distributions include all of the continuous and discrete random distributions. The perturbation function

```
newvalue(x)= equilikely(
floor(oldvalue(x)*0.6),
floor(oldvalue(x)*1.40))
```

is an example of a discrete distribution that perturbs a value by substituting an equilikely random value on the interval of 40% more and 40% less than the expected value. This function however leaves the possibility of returning `newvalue(x) = oldvalue(x)`. Conditions are placed in the code that executes this function to avoid this.

For example, if we wanted to change a's value to something close to what it has after this computation,

```
a = a + 1;
```

we would replace the original statement with the following code chunk:

```
a = a + 1;
a = newvalue(a).
```

The code for `newvalue()` would also be added somewhere into the program and would look like the following pseudo-code:

```
int newvalue(int a)
{
  counter = 1;
  oldvalue = a;

  do
  {
     a = equilikely( floor(oldvalue * 0.6),
                     floor(oldvalue * 1.4) );
     counter++;
  }
```

```
  while ( (a == oldvalue) && (counter < 100) );

  if ( (counter == 100) && (a == oldvalue) )
  {
    a = oldvalue - 1;
  }

  return (a);
}
```

(Note that 0.6 and 1.4 can be modified to however tight or loose of an interval as is desired. For example, 0.0001 and 10000 could be used to widen the interval of choices.)

Because this function could result in an infinite loop while trying to find a different value, a counter is added to ensure that after 100 attempts, the loop terminates and simply decreases the value of `a` by one. (We could have just as easily decided to program it to increase the value by one or even flip a coin as to which it does.)

Note that we can also use fault injection to modify the time at which code is executed by adding function calls that slow down the software. For example, in Ada, we can add a `delay(5)` statement to stop a process from executing for 5 milliseconds. And we can even simulate events such as the software's state, stored in memory, having its bits toggled due to radiation or other electromagnetic corruption. The `flipBit` function which will now be described provides this capability.

`flipBit`

The perturbation function `flipBit` toggles specific bits. The first argument to `flipBit` is the original integer value and the second argument is the bit to be toggled (we assume little-endian notation). The function `flipBit` is then written in C as follows and linked with the executable. Note that the ^ represents the XOR operation in C and the << operator represents a SHIFT-LEFT of y positions.

```
void flipBit(int *var, int y)
{
   *var = *var ^ (1 << y));
}
```

`flipBit` can serve as the underlying engine from which other perturbation function can be created. For example, to toggle two or more randomly selected bits in the integer, we can employ `flipNbits`:

```
void flipNbits(int *var, int n)
{
        int bits = 0;
        int bitPos = 1;
        int i,j,k;
        int xbit;
        for (i = 0; i < n; i++)
        {
           bits |= bitPos;
           bitPos <<= 1;
        }
        for (j = 0; j < sizeof(int) * 8; j++)
        {
           xbit = lrand48()
           if ((!!(bits & (1 << xbit))) !=
               (!!(bits & (1 << j))))
           {
              flipBit(&bits, xbit);
              flipBit(&bits, j);
           }
        }
        for (k = 0; k < sizeof(int) * 8; k++)
           if (bits & (1 << k))
               flipBit(var, k);
}
```

3.2 Fault Injection Security Tool

We will now discuss our Fault Injection Security Tool (FIST). The tool automates the analysis of security-critical software and requires program inputs, fault injection directives (meaning information about how to corrupt program states), and assertions written in C and C++ (that define when security of the software has been compromised). A schematic diagram of FIST is shown in Figure 1.

The fault injection engine provides a developer or analyst the ability to perturb program states randomly, append or truncate strings, attempt to overflow a buffer, and perform a number of other numerical fault injection functions. The security policy assertion component provides a developer or analyst the ability to code the security policy of the program under analysis as well as system security constraints.

Using FIST is a four step process: instrument, compile, execute, and analyze. The source code is instrumented with assertions and perturbation functions using a source code browser component. The browser tells the user all the legal points in the source where instrumentation can be attached. The user places instrumentation according to the desired analysis, then the instrumented code is compiled. Next, the instrumented program is executed repeatedly, once for each perturbation function that was encountered during an unperturbed run of the program. In each execution, only one location is perturbed. Any assertions that fire during the runs are noted. Relative security met-

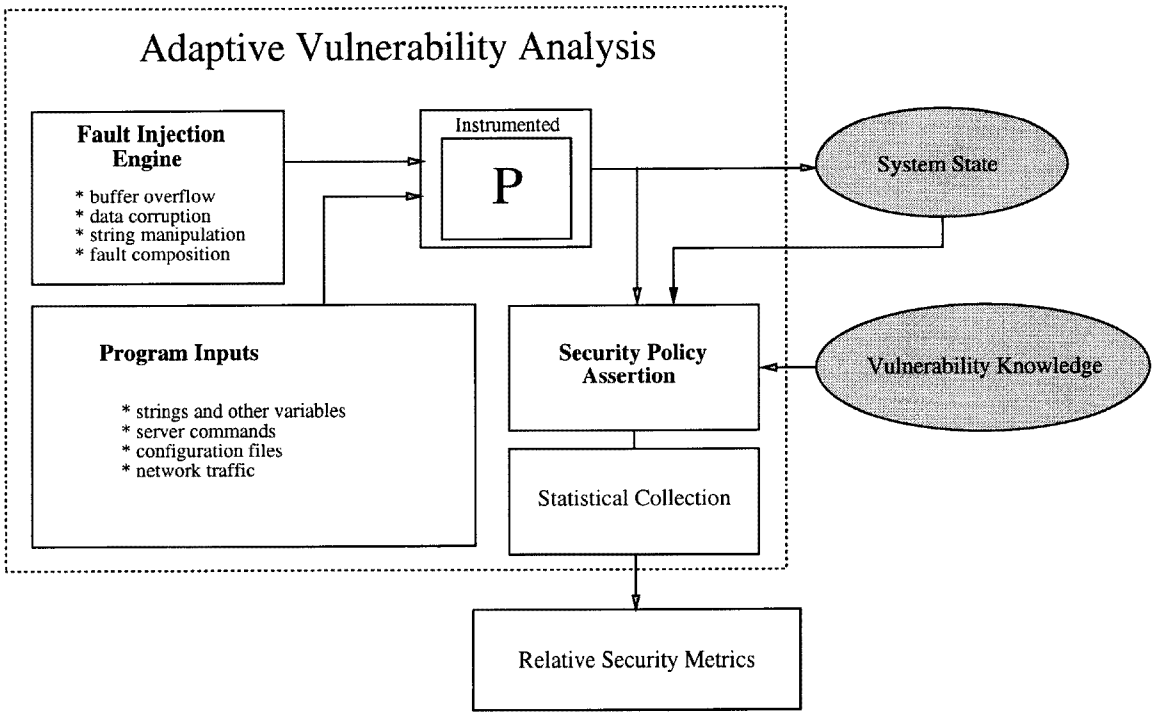

Figure 1: Overview of the Fault Injection Security Tool. A program, P, is instrumented with fault injection functions and assertions about its security policy (based on the vulnerability knowledge of the program). The program is exercised using program inputs. The security policy is dynamically evaluated using program and system states. If a security policy assertion is violated during the dynamic analysis, the specific input and fault injection function that triggered the violation is identified. Algorithm 1 is used to collect statistics about the vulnerability of the program to the perturbed states. One output from the analysis is the relative security metric $\hat{\psi}_{alPQ}$.

rics are accumulated for each program location that indicate the percentage of runs where a fault injection function at that location resulted in a security violation. The user can browse the result of the experiments using a results browser that links results to the original source code.

A fault injection engine has been implemented to support injection of anomalous states as well as specific exploits to test for vulnerability to known malicious threats during the execution of the program. Fault injection functions are instrumented by default in every viable program location to permit analysis of software flaws anywhere in the program source code. The reasoning is that without prior knowledge of where actual flaws exist, simulating their effects everywhere during automated analysis can identify which locations are most likely to impact security. Recall from the algorithm that program states are perturbed singly in each test run in order to assess the effect of a single flaw in a given location.

Fault injection is useful for simulating a variety of anomalous program behavior that would otherwise be very difficult, if not impossible, to simulate using standard testing. The main use of fault injection functions for vulnerability analysis is to determine where potential weaknesses exist in a software program that can be leveraged into security violations. Fault injection also reveals the relative importance of variables, statements, or whole functions on the output (and security) of a program. For example, perturbing the result of a display function may have little or no effect on the output of a program. On the other hand, perturbing the result of a function that parses user input, may well affect the output and perhaps even the security of the application. Finally, fault injection can be used to simulate malicious threats against a software application such as buffer overrun threats. We describe these uses of fault injection in the Section 3.3.

FIST includes numerous fault injection functions for all primitive data types ranging from simple

Boolean state flips, to string mangling, to "stack smashing" buffer overflow functions. These functions include the ability to corrupt Booleans, characters, strings, integers, and doubles. The Boolean perturbation function applies a logical negation operation to an unperturbed value. The character perturbation function returns a character randomly selected from the ASCII table. String perturbation functions provide the ability to truncate strings, concatenate a random string, concatenate a fixed string, generate a new string of random characters, and replace strings with a string randomly selected from a file. In addition to simple fault injection functions, FIST supports composition of fault injection functions from a combination of selected basic fault injection functions. For example, a user can append a fixed string with a random character fault perturbation, thus building a new fault injection function.

The buffer overflow function overwrites the return address of the stack frame in which the buffer is allocated with the address of the buffer itself. By tracing the frame pointer back through the stack, the fault injection function is able to determine where to overwrite the return address. The opcodes for machine instructions are written into the buffer being perturbed. Eventually, the activation record containing the modified return address will be popped off the program stack and the program will jump to the machine instructions embedded by the fault injection function. These instructions will be executed as if they were a part of the normal operation of the program. Because different platforms implement different forms of program stacks, the buffer overflow fault injection functions are platform-dependent. Linux x86 and Sparc are the two platforms currently supported.

Unsafe languages such as C make buffer overflow attacks possible because of input functions such as `gets`, `strcat`, and `strcpy` that do not check the length of the buffer into which input is being copied. If the length of the input is greater than the length of the buffer into which it is being copied, then a buffer overflow can result. Safe programming practices that read in constrained input can prevent a vast majority of buffer overflow attacks. However, many security-critical programs in the field today do not employ these safe programming practices. In addition, many of these programs are still coded in commercial software development labs in unsafe languages today.

FIST detects the potential for buffer overflow attacks to be successful regardless of how the input is read. Searching for unsafe functions such as `strcat` and `strcpy` is one technique for detecting potential problems; however, it is insufficient by itself. Programmers often write their own dangerous input functions that read in unconstrained input. FIST attempts to overflow buffers regardless of whether the buffer is used in a known dangerous function or is used in a custom-written input function. Furthermore, FIST can overflow buffers for variables that are not pushed on the stack. While this type of perturbation may not result in the execution of arbitrary program code, it may have side effects that compromise program security by corrupting other variables used for access/privilege decisions. If the fault injection function results in a security policy breach, the programmer must either ensure that the vulnerable buffers cannot be overflowed from user input or use safe programming practices to ensure that the buffer overflow cannot occur. Once patched, FIST can be re-run to determine if the patch is resilient to attack.

As an alternative to the source-code–based analysis approach, StackGuard, a `gcc` compiler variant for Linux developed by the Oregon Graduate Institute, attempts to protect buffers from stack smashing attacks by aborting the program if the return address pushed on the stack is overwritten [2]. Stack Guard will not protect programs against all buffer overflow attacks, but can prevent stack smashing attacks from running arbitrary code embedded in user input. For example, buffer overflow attacks that overwrite local variables that were never intended to be user changeable can result in security violations not prevented by StackGuard [1].

The Fuzz tool [4] can be used to overflow buffers, too, but with inconclusive results. Because the input is randomly generated, the vulnerability of the program to executing user-defined code cannot be assessed. FIST implements specific fault injection functions that determine the program's vulnerability to specially-crafted buffer overflow attacks.

FIST integrates with the normal build process of the application under analysis. Any source file that is compiled using the FIST pre-processor at build time is instrumented. Libraries can be instrumented using FIST and then linked to applications, but only if the source code for the library is available. Uninstrumented libraries can also be linked to instrumented applications.

The security-policy–monitoring component of FIST allows users to specify what constitutes a security violation for the software application under analysis. Using assertions to encode this policy, the policy is monitored during the dynamic analysis to determine if it has been violated. The nature of violations will vary

from application to application, and the types of violations the user will seek to detect will generally be dependent on both the input to the program and fault injection functions. As a result, the analyst must determine the security policy for the program being analyzed. A number of pre-defined assertion functions have been developed from which a user can specify the security violations for internal program variables, environment variables, and external system states.

Perhaps the broadest assertion function FIST provides allows the user to develop any expression in C to represent a violation assertion. This expression is evaluated during execution to determine if a violation has occurred. If the result of the expression is non-zero, then the violation is assumed to have occurred. This function has been developed for a sophisticated user who does not want to be constrained by the pre-packaged functions provided in the tool. Assertion functions are placed at locations in the source code during the instrumentation step. FIST also provides a mechanism for external assertion monitoring.

The external assertion monitor runs in parallel with the instrumented program and uses a subset of the built-in assertion functions. It is able to monitor files on the system, checking for modifications and/or accesses. For the buffer overflow functions, FIST checks for side effects of the `mycmd` program. The assertion is coded such that a file called `touch.out` should not be modified during the execution of the instrumented program. This assertion will be violated if the buffer overflow succeeds and the `mycmd` program is executed, which in turn will open `touch.out` and modify it. So when checking for buffer overflows, the security policy is simple: `touch.out` should never be modified.

3.3 Case studies of security-critical software

FIST analysis was performed on five different network services. Network service daemons are interesting case studies from a security standpoint because they provide services to untrusted users. Most network daemons typically allow connections from anywhere on the Internet, leaving them vulnerable to attack from malicious users anywhere. Network daemons sometimes run with super-user, or `root`, privilege levels in order to bind to sockets on reserved ports, or to navigate the entire file system without being denied access. Successfully exploiting a weakness in a daemon running with high privileges could allow the attacker complete access to the server. Therefore, it is imperative that network daemons be free from security-related flaws that could permit untrusted users access to high privilege accounts on the server.

The programs examined were NCSA `httpd` version 1.5.2.a, the Washington University `wu-ftpd` version 2.4, `kfingerd` version 0.07, the Samba daemon version 1.9.17p3, and `pop3d` version 1.005h. The source code for these programs is publicly available on the Internet. Samba, `httpd`, and `wu-ftpd` are popular programs and can be found running on many sites on the Internet. The analysis of those programs was performed on a Sparc machine running SunOS 4.1.3_U. The other programs, `pop3d` and `kfingerd`, are Linux programs found in public repositories for Linux source code on the Internet. The analysis of those programs was performed on a Linux 2.0.0 kernel. The programs were instrumented with both simple fault injection functions as well as the buffer overflow functions where applicable.

A summary of results from the analysis is shown in Table 1. The table shows the total number of instrumented locations together with the number of simple perturbations and buffer overflow perturbations that resulted in security violations. The last column shows the percentage of the functions in the source code that were executed as a result of the test cases employed. Higher coverage results may result in more potential security hazards flushed out through the analysis. The results should not be interpreted to mean that the locations identified in the analysis are necessarily exploitable, only that they require closer examination from the software's developers to determine if they can be exploited from input and whether fault-tolerant mechanisms should be employed. It is worth mentioning, however, that one of the potential buffer overflow vulnerabilities found in `wu-ftpd v2.4` and published in [6] was later reported in CERT Coordination Center, Pittsburgh, PA, CERT Advisory CA-99-03, "FTP Buffer Overflows" (see www.cert.org).

4 Interface Propagation Analysis

Much of our research during the past 4 years has been geared toward increasing the observability of large-scale information systems. The main "ilities" that our work has addressed are security and safety.

The premise of our approach is as follows: since it is rarely possible to guarantee "correct" behavior at the system or component level, we should instead focus on guaranteeing levels of "acceptable" behavior. In essence, we should work to thwart system level failures that are the most undesirable and ignore the rest.

Our approach is simple. Start from an assumption about the worst behaviors from a component and observe how that will affect the full system. If the effect is negligible, ignore the component. If the impact is

Program	Instrumented Locations	Successful Simple Perturbations	Successful Buffer Overflows	Function Coverage
Samba v1.9.17p3	1264	12	15	45.5%
NCSA `httpd v1.5.2a`	463	27	3	40.14%
`wu-ftpd v2.4`	476	11	3	58.62%
`pop3d v1.005h`	73	2	1	63.64%
`kfingerd v0.07`	146	12	5	38.1%

Table 1: **Results from FIST analysis of network daemons.**

large, it is clear that the component is one that needs scrutiny. The bottom line is that we do not care how poorly subsystems behave as long as their behaviors do not jeopardize the integrity of the full system.

Given that resources are always too few, this perspective provides an intelligent way to allocate component testing resources, i.e., to components that have demonstrated a capacity to cause undesirable, system-wide problems.

The approach we have developed is termed Interface Propagation Analysis (IPA). IPA is a fault injection-based technique that simulates component and subsystem failures.

IPA is normally applied once the system is completed. IPA can also be applied before a component is built, provided there exists a specification for what the component is expected to do. (Components that do not yet exist are termed "phantom components"). And finally, IPA can also be used to test the robustness of individual components.

IPA is made of two software fault injection algorithms: "Propagation From" (PF) and "Propagation Across" (PA). PF corrupts the data exiting a real component (or phantom component) and observes what it does to the remainder of the system (*i.e.*, what type of system failures ensue, if any). PF can also observe whether other subsystems fail and how. Thus, PF is an advanced testing technique that provides the raw information needed to measure the semantic interactions between components in order to measure their tolerance to one another.

PA corrupts the data entering a component. This process simulates the failure of system components that feed information into the component in order to see how it reacts. These simulated failures mimic human operator errors, failures from hardware devices, or failures from other software subsystems. After the component under analysis is forced to receive corrupt input, PA observes whether the component chokes on the bad data and fails. Note that PA is very similar to PF. The only difference is scale: PA is focused on standalone components and PF is focused on component interactions.

5 Conclusions

In this paper, we described the use of an off-nominal testing approach — fault injection analysis — to test the survivability of an information system to three different types of events:

- software flaws in program source code,
- malicious attacks against programs,
- anomalous behavior from third party software.

Source-code–based fault injection analysis can be applied either to open source software after software is released or to software during development by software vendors. The earlier in the software lifecycle off-nominal testing techniques are used, the cheaper the cost to find and correct bugs. The Fault Injection Security Tool supports testing of the first two scenarios above: simulation of software flaws and malicious attacks against programs. The tool was applied to several commonly deployed open source systems. Even with the low levels of code coverage, several potential security-related hazards were demonstrated, one of which was later independently found and reported to the CERT CC.

The third scenario is becoming increasingly important. Software developed and released today is heavily dependent on third party or COTS software. Anomalous behavior from third party software can result in system-wide failure. Interface Propagation Analysis addresses the survivability of a system composed of custom and third-party components by using fault injection analysis at component interfaces. The fault injection analysis can determine the effect of failing or anomalous behavior of third party software on system survivability. This technology is a key step from moving from "testing in the small" to "testing in the large".

5.1 Acknowledgements

Earlier versions of this paper can be found at:

1. A. Ghosh and J. Voas. "Innoculating Software for Survivability," *Communications of the ACM*, 42(7):38-44, July, 1999.

2. J. Voas and A. Ghosh. "Software Fault Injection for Survivability", In *Proc. of the DARPA Information Survivability Conference and Exposition*, January, 2000.

References

[1] S. Bellovin. Re: Stackguard: Automatic protection from stack-smashing attack. Online. Bugtraq archives. See http://www.geekgirl.com/bugtraq/1997_4/0514.html, December 19 1997.

[2] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, San Antonio, TX, January 1998.

[3] M. Daran and P. Thevenod-Posse. Software error analysis: A real case study involving real faults and mutations. In *Proceedings of the 1996 Int'l Symp. on Software Testing and Analysis*, pages 158–171. ACM Press, January 1996.

[4] B.P. MILLER ET AL. Fuzz revisted: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin, Computer Sciences Dept, November 1995.

[5] R. BUTLER AND G. FINELLI. The infeasibility of experimental quantification of life-critical software reliability. In *Proceedings of SIGSOFT '91: Software for Critical Systems*, pages 66–76, New Orleans, LA, December 1991.

[6] A.K. Ghosh, T. O'Connor, and G. McGraw. An automated approach for identifying potential vulnerabilities in software. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 104–114, Oakland, CA, May 3-6 1998.

[7] J. VOAS AND G. MCGRAW. *Software Fault Injection: Inoculating Programs Against Errors.* John Wiley and Sons, New York, 1998.

[8] J. D. MUSA, A. IANNINO, AND K. OKUMOTO. *Software Reliability Measurement Prediction Application.* McGraw-Hill, 1987. ISBN 0-07-044093-X.

[9] R. A. DEMILLO, R. J. LIPTON, AND F. G. SAYWARD. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.